LA-UR- 02-6057

Approved for public release;
distribution is unlimited.

Title:

An Automated Regression Testing and Verification System for ParSim (Parallel Architecture Simulation Framework)

Author(s):

Nick Moss

Submitted to:

world wide web

Los Alamos
NATIONAL LABORATORY

An Automated Regression Testing and Verification System for ParSim (Parallel Architecture Simulation Framework)[*]

Nick Moss - nickm@lanl.gov
Modeling, Algorithms, and Informatics Group
Los Alamos National Laboratory

November 4, 2002

Abstract

The Parallel Architecture Simulation Framework (ParSim), like any other software of large scale, requires considerable efforts in verification and validation. ParSim, built on Dartmouth Scalable Simulation Framework (DaSSF), which in turn uses Domain Modeling Language (DML) facilitates a solution by separating models and runtime parameters from implementation. Through this design it is possible to simulate a wide variety of architectures while sharing the same code base. Test cases are simply an extension of this separation—we validate output from a set of predefined models and runtime parameters and throughout changes to the code base, test to ensure that their output remains consistent, either exactly or according to some criterion. This document discusses the design and usage of a modular system that automates this process and provides a flexible framework for specification and execution of test procedures.

1 Introduction

ParSim is an ongoing project to produce high fidelity simulations[1] of ASCI/Extreme-scale (more than one thousand processors) parallel architectures with an end goal of providing predictive capability in the evaluation of existing and hypothetical architectures, and the performance of programs run on them. We model machines as constructed from a set of basic components including switches, nodes, and network interface cards. The components approximate their real counterparts in varying levels of detail and include parameters for bandwidth, transmission delay, and others in the case of switches; for nodes and network interface cards possible parameters include packet size, routing methods, and others which determine messaging behavior. The underlying network is modeled at the packet level and based on the Quadrics Elan 3 protocol. Message and packet transmission through the simulated network, and the trace events generated through this process, provide useful data for analyzing performance. Another approach, using ParSim's direct execution feature, allows any MPI program to execute on the simulated machine as if it were executed on the actual machine. Because performance is highly dependent on applications and how well

[*]This work was funded by the Department of Energy under ASCI Discom2

they run in a parallel environment, direct execution is a powerful method of analysis in the evaluation of hypothetical architectures.

1.1 Framework and Implementation

ParSim is built on Darthmouth Scalable Simulation Framework[2], a C++ implementation of the Scalable Simulation Framework specification (SSF). DaSSF is well suited for discrete event simulations of large scale such as those we are interested in studying. The SSF specification originally envisioned simulating the Internet and other massive dynamic networks, and for ParSim, where the network is a central component, the SSF API is a natural fit. ParSim component classes define network components, routing methods, messages, packets and other constructs. They are implemented in C++ and derived from the base classes defined by the SSF specification: *Entity*, *Event*, *Process*, *Input Channel*, and *Output Channel*.

1.2 DML

A model may be constructed by instantiating components within C++ code, but DaSSF provides a better approach using the Domain Modeling Language (DML), which isolates model specification from implementation. DML is a generic syntax which supports recursive-like nesting of submodels, or subsections of DML code, and relatively convenient specification of large structures. DaSSF adapts the generic DML syntax, adding various extensions and defining its semantics. DaSSF uses DML for specification of runtime parameters `runtime.dml`, runtime architectures `machine.dml`, and most significantly, model construction `model.dml`.

2 Overview

ParSim is capable of simulating a diverse range of architectures. The architectures may be large and complex but are all composed from a common set of components. Different configurations and paramerization options exist for nodes, network interface cards, and switches, and other ParSim components, but are derived from a common implementation. It is important to ensure that, in the course of development, as we make changes that can have wide-ranging effects and that seem to have the desired effect in some limited scope, they not introduce irregularities in other systems. For this reason, and for the size and complexity in general of the ParSim system, verification is vital.

The verification process includes a large set of test models or test cases. We validate the output of these models, in some cases by hand checking or through some automated process. After changes to implementation code we recompile ParSim and execute with these test models, checking their output against the validated output.

3 Test Program Design

The modularity provided to ParSim through DaSSF's use of DML enables an efficient solution to the implementation of a automated system for execution of test processes. The `model.dml` file is a complete ParSim model which contains a series of DML statements that parameterize basic components, Node, NIC, and Switch, and their interconnections. The example below illustrates a trivial model

```
 ENTITY [
    INSTANCEOF "Node"
    PARAMS [
      INT    0
    ]
]

 ENTITY [
    INSTANCEOF "NIC"
    PARAMS [
      INT    1
    ]
]

 ENTITY [
    INSTANCEOF "Switch"
    PARAMS [
      INT    2
    ]
]

MAP [FROM 0(NETOUT) TO 1(NETIN) DELAY 1]
MAP [FROM 1(NETOUT) TO 0(NETIN) DELAY 1]
MAP [FROM 1(NETOUT) TO 2(LINKIN0) DELAY 1]
MAP [FROM 2(LINKOUT0) TO 1(NETIN) DELAY 1]
```

The `runtime.dml` file contains runtime parameters and globals such as simulation interval, output options, and other ParSim-specific settings. For example

```
EXECUTABLE      "parsim3"
MODEL           "model.dml"
MACHINE         "machine.dml"
STARTTIME       0
ENDTIME         2e+08
ENVIRONMENT [
  LOG_STYLE     "IMMEDIATE"
  LOG_MASK      "ERROR WARNING INFO"
  DATA_STYLE    "AT WRAPUP"
  DATA_MASK     "INFO"
  EPILOG_STYLE  "IMMEDIATE"
  EPILOG_FILE   "output.elg"
  WORKLOAD      "TestWorkload"
  NETWORK       "MFQuadrics"
  FIRST_NODE    0
  FIRST_NIC     100000
  FIRST_SWITCH  200000
  MPI_COMM_SIZE "64"
]
DATAFILE        "output"
```

3.1 Test Cases

A *test case* is simply a model and a collection of runtime parameters, provided in the `model.dml` and `runtime.dml` files. The test suite contains a collection of test cases, some simple and others complex, which simulate important mechanisms and behavior in ParSim systems and whose output is included in the verification process.

3.2 Execution Output and Source Reference Data

Execution of an input model with a given set of runtime parameters produces

```
ID      Time    Location        Action  Source  Target  Size    Type
0:1     10      0       MessageSent     0       1       100     Null
0:1     13      10000   MessageReceived 0       1       100     Null
0:1:1   526.513 10000   PacketSent      0       1       100     Head
0:1:1   543.513 20000   PacketReceived  0       1       100     Head
0:1:1   562.513 20000   PacketSent      0       Null    100     Head
0:1:1   575.513 10001   PacketReceived  0       Null    100     Head
0:1:1   686.114 10000   PacketSent      0       Null    0       AckNow
0:1:1   703.114 20000   PacketReceived  0       Null    0       AckNow
...
```

whose output traces message and packet transmission through the simulated network for a given interval. We store execution output from the validated models to use as *source reference data* in the verification process.

3.3 Test Modules

Test modules at this level are a sort of abstract entity. We pass as input the execution output from the current test model and the source reference data relevant to that test case. The test module may perform any sort of verification in comparing source reference data to execution data and outputs pass or fail and additional information, warnings, reasons for failure, etc.

4 Test Program Organization

The test program files are included in the ParSim source directory and organized as:

```
rtest
rtest.config
config/out
config/models
config/runtime
config/source
config/test_modules
```

where `rtest` is typically executed from the ParSim source directory and `config` contains subdirectories for execution ouput, test models, runtime parameters test models, source reference output, and test modules, respectively.

The `rtest` program is responsible for coordinating the regression test process and determines which models are to be included in the test group, calling `make` to compile ParSim with a particular test model and runtime parameters, execution, collection of output, and finally, calling testing modules to peform verification procedures.

4.1 Configuration

`rtest.config` is a DML-like, whitespace-delimited key/value configuration file. In the following example

```
MODELS testA, testB, testC, testD
MODELS_DML_PATH src/dml
MAKE_CLEAN 0
```

`MODELS_DML_PATH` sets the directory containing the models to be included in the tests. `MODELS` is a comma-delimited list of test cases that correspond the models in `config/models` `MAKE_CLEAN` is a boolean flag indicating whether or not to execute `make clean` prior to compilation.

4.2 Runtime Parameters

Runtime parameters may be applied to a general set of test models or individually. For example, `default.runtime.dml` is applied to all test cases, with the exception of one particular test model, testA.dml that requires different parameters included in `config/runtime/testA.runtime.dml`. The system uses a systematic naming convention to determine runtime paramters, source reference data and test modules, the details of this are outlined in next section.

4.3 File Naming Conventions

Appropriate test modules, runtime parameters, and source reference data are selected according the following naming convention

<default or all or model_name>.module_name.context

For example, possible names for runtime parameters are

```
default.runtime.dml
all.runtime.dml
testA.runtime.dml
```

where `default` specifies the default runtime paramaters for a test case unless bypassed by `all` or when runtime parameters exist for a specific model. The `all` prefix is used in a similar way but is never overriden.

4.4 Test Modules

The test program does not define or restrict the exact operation of the test module or `test script` but provides a standard mechanism for their execution. `rtest` determines which test scripts to use based on the named test models in the configuration according the naming convention outlined above. `rtest` executes matching test scripts and passes them the source reference data and output from the current execution. Output from the script is expected to be 0 in cases of ”pass”, with any number of arbitrary results printed to *stdout*. Test module output is stored separately for each test case along with any other output from their execution. The script may written in any language and can perform any sort of testing so long as it conforms to the input and pass/non-pass output interface expected by `rtest`.

Following the naming convention ensures that the test script will be called in the correct context. The script is called passing the two inputs as

```
test_script <source_file> <output_file>
```

where *source_file* is the predefined source data file supplied from `config/source` and *output_file* contains output data from the current test case. Minimally, the script processes *output_file* and in most cases also examines *source_file*. Through this mechanism it possible to implement a wide variety of comparisons and constraints verification tests.

4.5 Test Module Implementation

The example below implements a simple *diff* test module, written in Perl which uses the standard UNIX `diff`

```
#!/usr/bin/perl

my( $source, $out ) = @ARGV;

my $output = `diff -bw --minimal $source $out`;
my $return_code = $?;
print $output;
if( $return_code ){
    exit( 1 );
}
```

4.6 Program Output

Execution of `rtest` on two test modules testA and testB produces

```
building testA
finished building testA
finished executing parsim3-X86-LINUX
all.diff.test: PASS.

building testB
finished building testB
finished executing parsim3-X86-LINUX
all.diff.test: FAILED. Details in: config/out/testS.all.diff.test

pass = 1
fail = 1
```

At the bottom, a count states the total number of test cases (which includes net result for all test modules) that passed or failed with details for each test. For testB, as noted, the output of diff (generated from the print statement in the script above is recorded in `config/out/testS.all.diff.test`)

References

[1] The *à la carte* Project. *Design and Implementation of Low- and Medium-Fidelity Network Simulations of a 30-TeraOPS System.* Los Alamos National Laboratory, April 2002.

[2] Jason Liu and David M. Nicol. *Dartmouth Scalable Simulation Framework User's Manual.* Dartmouth College, Department of Computer Science, August 2001.